**Spring Security**

**1.Importance of Security in Software Applications**

1. **Protect Sensitive Data**

* **Why it matters**: Applications often handle sensitive data like personal information, financial details, and business secrets.
* **Risk**: Data breaches can lead to identity theft, financial losses, and regulatory fines.
* **Example**: An e-commerce app must secure customer payment details and personal information to comply with standards like PCI DSS.

2. **Prevent Unauthorized Access**

* **Why it matters**: Without proper security, unauthorized users may gain access to sensitive areas of the application.
* **Risk**: Leads to compromised data integrity or misuse of application functionality.
* **Example**: Implementing multi-factor authentication (MFA) prevents attackers from hijacking user accounts.

3. **Maintain User Trust**

* **Why it matters**: Security breaches damage an organization’s reputation.
* Risk: Users may stop using the application due to lack of confidence in its security.
* **Example**: A social media platform must ensure security to retain its user base and maintain trust.

4. **Comply with Regulations and Standards**

* **Why it matters**: Applications must comply with legal requirements like GDPR, HIPAA, or CCPA.
* **Risk**: Non-compliance leads to hefty fines and legal consequences.
* **Example**: Healthcare apps must encrypt patient records to meet HIPAA regulations.

5**. Safeguard Against Financial Loss**

* **Why it matters**: Security vulnerabilities can result in financial losses due to fraud, ransom demands, or system downtime.
* **Risk**: Costs of handling security breaches, including compensation and recovery expenses.
* **Example**: A banking application must use strong encryption to prevent financial fraud.

6. **Ensure Application Availability**

* **Why it matters**: Attacks like Distributed Denial of Service (DDoS) can disrupt application services.
* **Risk**: Downtime leads to loss of revenue and customer dissatisfaction.
* **Example**: Cloud-hosted applications should implement firewalls and monitoring to mitigate such risks.

7. **Prevent Intellectual Property Theft**

* **Why it matters**: Applications often contain proprietary code or business logic.
* **Risk**: Hackers can reverse-engineer insecure applications or steal codebases.
* **Example**: A SaaS company must secure its APIs to prevent unauthorized access to core functionalities.

8**. Avoid Internal Threats**

* **Why it matters**: Security isn’t just about external threats—disgruntled employees can also pose risks.
* Risk: Insider threats can lead to intentional data breaches or system sabotage.
* **Example**: Implementing role-based access control (RBAC) limits internal users’ privileges.

9**. Support Business Growth**

* **Why it matters**: Secure applications attract more users and expand markets.
* **Risk**: Lack of security limits the ability to onboard enterprise clients who demand high-security standards.
* **Example**: A secure application can confidently scale to handle a larger customer base.

**10.Application Security: Interview Perspective**

**Common Questions and How to Answer**

**10.1.0 Why is application security important**?

* Sample Answer: "Application security is crucial to protect sensitive data, prevent unauthorized access, ensure compliance with regulations, and maintain user trust. It also safeguards businesses against financial losses, reputational damage, and operational downtime.
  + 1. **What are some key principles of secure application design?**
* **Least Privilege**: Users and components should have the minimum access required.
* **Data Encryption**: Encrypt sensitive data in transit and at rest.
* **Authentication and Authorization**: Use strong identity verification mechanisms like MFA.
* **Input Validation**: Sanitize inputs to prevent injection attacks.
* **Logging and Monitoring**: Track and respond to suspicious activity.
  + 1. **What are common security vulnerabilities in applications?**
* **SQL Injection**: Exploiting input fields to execute malicious SQL queries.
* **ross-Site Scripting (XSS):** Injecting scripts into web pages viewed by others.
* **Cross-Site Request Forgery (CSRF):** Forcing users to perform unwanted actions.
* **Weak Authentication**: Poor password policies or lack of MFA.
* **Unsecured APIs**: Failing to protect API endpoints from unauthorized access.

10**.1.3 How do you ensure security in a software application?**

* Conduct threat modeling during the design phase.
* Perform regular security testing (e.g., penetration testing, code reviews).
* Use security frameworks like Spring Security or OWASP’s recommendations.
* Implement robust logging and monitoring solutions.
  + 1. **How do you handle sensitive data in an application?**
* Use encryption standards like AES-256 for data at rest and TLS for data in transit.
* Store sensitive data securely using tools like AWS KMS or Azure Key Vault.
* Follow tokenization or pseudonymization to reduce exposure of raw data.

**10.1.5 Key Takeaways for Notes**

* Security is a non-negotiable part of modern applications.
* Key focus areas include data protection, user trust, and regulatory compliance.
* Knowledge of common threats (e.g., OWASP Top 10) is essential.
* Highlight real-world examples and tools/frameworks during interviews.

**2.OWASP Top 10**

**2.1 Broken Access Control**

**What It Is**: Failure to enforce proper restrictions on what users can do or access.

**Examples**:

* Users accessing other users' accounts by modifying URLs:  
  Changing https://example.com/user/123 to https://example.com/user/124 allows unauthorized access to another user's account.
* APIs allowing unauthorized deletion or modification of data:  
  A user without admin privileges sends a request to delete data:

DELETE /api/admin/delete-user/456

If no access control checks exist, the user can delete data.

**Mitigation**:

* Enforce role-based access control.
* Implement checks to verify user permissions before processing requests.
* Regularly test for access control flaws.

2. 2 **Cryptographic Failures**

**What It Is:** Weak or missing encryption of sensitive data.

**Examples:**

* Storing passwords in plain text:  
  If the database is breached, all passwords are exposed.

**User: admin, Password: password123**

* Transmitting sensitive data (e.g., credit card info) over HTTP instead of HTTPS, exposing it to interception.

**Mitigation:**

* Use strong hashing algorithms like bcrypt for passwords.
* Always transmit data over HTTPS.
* Encrypt sensitive data at rest using AES (Advanced Encryption Standard).

2.3. **Injection**

**What It Is**: Injection occurs when untrusted input is sent to a system, causing unintended commands to be executed.

**Examples**:

* **SQL Injection**:  
  A login form vulnerable to SQL injection:

SELECT \* FROM users WHERE username = 'admin' AND password = ' ' OR 1=1 ';

This allows the attacker to bypass authentication.

* **Command Injection**:

An attacker enters ; rm -rf / in a vulnerable field, deleting files on the server.

**Mitigation**:

* Use parameterized queries or an ORM (Object-Relational Mapper) like Hibernate.
* Sanitize and validate all user inputs.

**2.4. Insecure Design**

**What It Is**: Applications designed without security in mind.

**Examples**:

* A password reset feature that sends a reset link without verifying the user’s identity.
* Storing user credentials in session variables without encryption.

**Mitigation**:

* Use secure design patterns such as enforcing strong authentication mechanisms.
* Threat modeling during the design phase.

**2.5 Security Misconfiguration**

**What It Is**: Misconfigured settings or default configurations leading to vulnerabilities.

**Examples**:

* Leaving the default admin credentials like admin/admin unchanged.
* Enabling detailed error messages that expose stack traces, helping attackers.

**Mitigation**:

* Change default credentials and disable unnecessary services.
* Restrict stack traces and error messages to internal logs.
* Automate security configurations using tools like **Ansible**.

**2.6 Vulnerable and Outdated Components**

**What It Is**: Using outdated libraries or software with known vulnerabilities.

**Examples**:

* Using an old version of **Log4j** that allows attackers to execute malicious code remotely.
* Running an outdated database system susceptible to SQL injection.

**Mitigation**:

* Regularly update dependencies and use tools like **Dependabot**.
* Subscribe to vulnerability alerts and test components for security flaws.

2.7 **Identification and Authentication Failures**

**What It Is**: Weak or mismanaged user authentication mechanisms.

**Examples**:

* Allowing weak passwords like "password123".
* Not implementing account lockouts for multiple failed login attempts, enabling brute-force attacks.

**Mitigation**:

* Enforce strong password policies.
* Implement rate-limiting and account lockout mechanisms.
* Use multi-factor authentication (MFA).

**2.8 Software and Data Integrity Failures**

**What It Is**: Failure to ensure that software and data are trustworthy.

**Examples**:

* Allowing unsigned or unauthenticated software updates, which attackers can replace with malicious versions.
* Manipulating JSON payloads during data transfers due to lack of integrity verification.

**Mitigation**:

* Use digital signatures for software updates.
* Implement hashing to verify the integrity of data.
* Secure the CI/CD pipeline with authentication and validation checks.

**2.9 Security Logging and Monitoring Failures**

**What It Is**: Inadequate logging and monitoring, leading to delayed or missed detection of attacks.

**Examples**:

* A brute-force login attack goes unnoticed because failed login attempts are not logged.
* No alerts generated when a database query retrieves millions of records unexpectedly.

**Mitigation**:

* Implement centralized logging systems like **ELK Stack**.
* Monitor logs for suspicious activities and configure automated alerts.

**2.10 Server-Side Request Forgery (SSRF)**

**What It Is**: SSRF occurs when an application fetches remote resources without properly validating user-supplied URLs.

**Examples**:

* An attacker sends this URL to your server

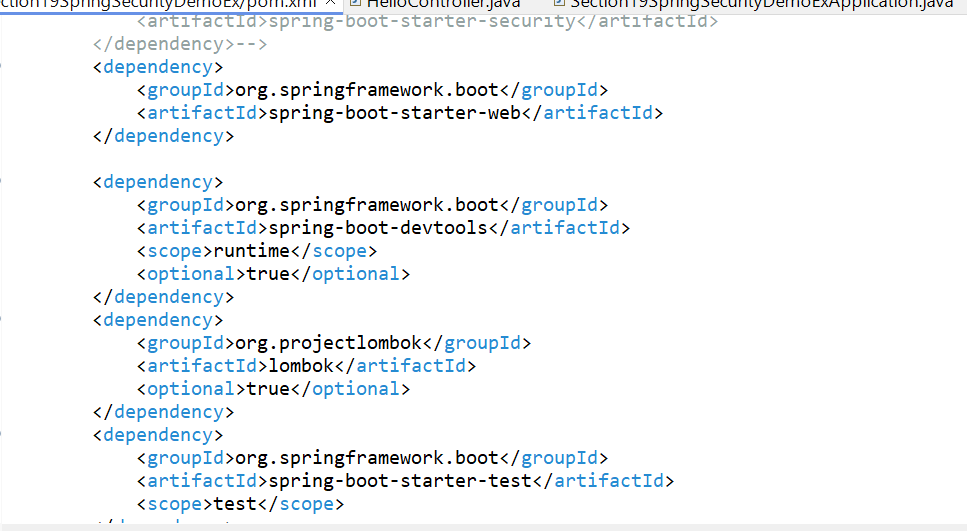
<https://yourserver.com/fetch?url=http://internal-server.local>

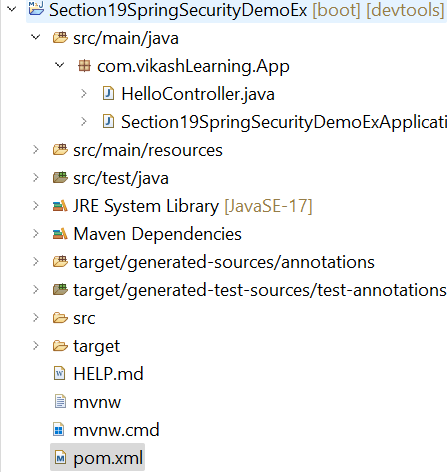
Your server unknowingly fetches internal resources and exposes sensitive data.

**Mitigation**:

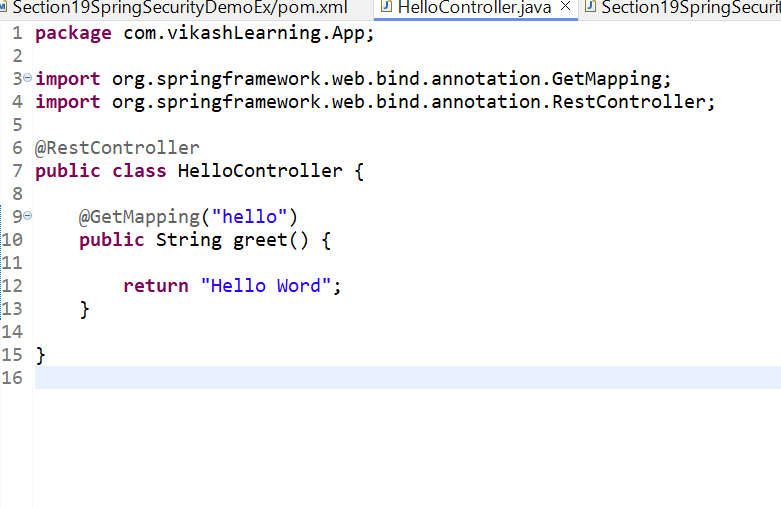
* Validate and sanitize all user-supplied URLs.
* Use a whitelist of allowed domains.
* Disable unnecessary network access for server components.

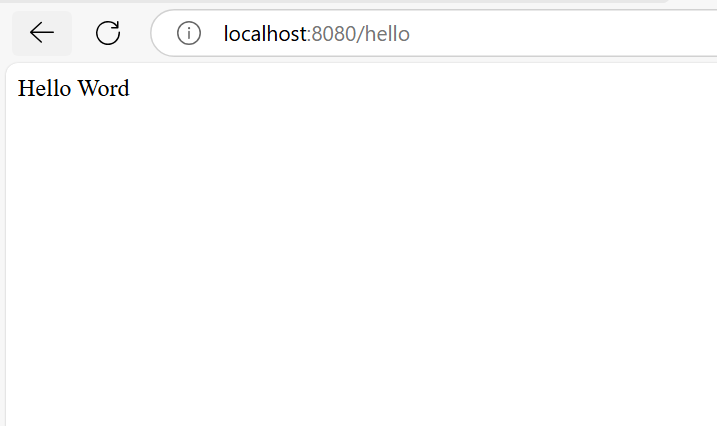
**3.Creating a Spring Security Project**





HelloController.java





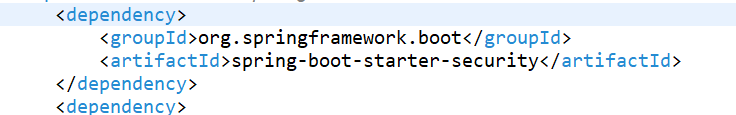
**4.default login form**

**Features of the Default Login Form:**

1. **Auto-Generated**: No additional configuration is required; it's provided out-of-the-box when Spring Security is added to your project.
2. **Username and Password Fields**: The form includes fields for entering a username and password.
3. **Error Handling**: Displays error messages if:
   * Login fails due to incorrect credentials.
   * The session is expired or invalid.
4. **Logout Functionality**: Includes a built-in logout endpoint (/logout).

**Steps to Enable the Default Login Form**

1. **Add Spring Security Dependency**: Add the following dependency in pom.xml for a Maven project:



1. **Run the Application**:

* When you start the Spring Boot application, the /login endpoint becomes available.
* Access the default login form by navigating to /login.

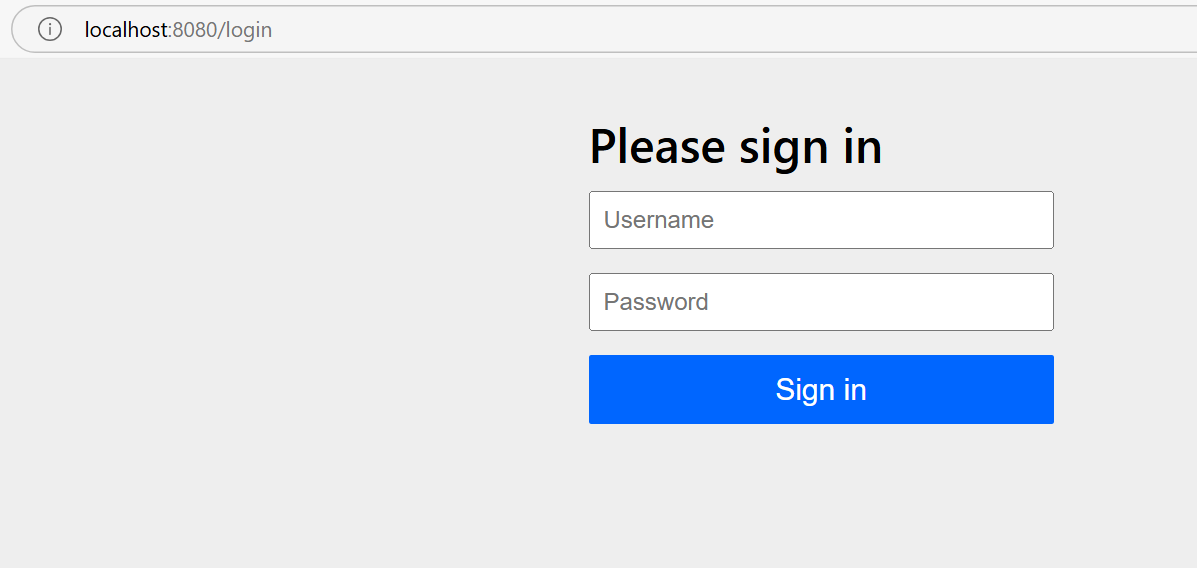
1. **Default User Credentials**:

* Spring Security auto-generates a user with the username user and a random password, logged in the console:

Using generated security password: 1e5f0c3a-9a2d-45ab-b09b-12d5d1a2345e

1. **Controller and View Not Required**:

* You don't need to create a controller or an HTML login page manually.



Enter username: user (bydefault) and password: generated automatically u can see in console

**5.Spring Security Filters**

**What is a Filter in Spring Security?**

A **filter** is a component that intercepts incoming HTTP requests and outgoing HTTP responses. In Spring Security, filters are used to enforce security policies like authentication and authorization before the request reaches the controller.

Spring Security uses a chain of filters known as the **Security Filter Chain**. Here are some of the primary filters:

**1.UsernamePasswordAuthenticationFilter**

* Handles form-based login (default /login).
* Extracts username and password from the request and delegates authentication to the AuthenticationManager.

**2.BasicAuthenticationFilter**

* Handles HTTP Basic authentication by extracting credentials from the Authorization header.

**3.BearerTokenAuthenticationFilter**

* Handles authentication for applications using OAuth 2.0 Bearer Tokens (e.g., JWT).

**4.CsrfFilter**

* Protects against Cross-Site Request Forgery (CSRF) attacks.
* Ensures requests include a valid CSRF token.

**5.ExceptionTranslationFilter**

* Catches security exceptions and sends appropriate responses (e.g., 401 Unauthorized).

**6.SecurityContextPersistenceFilter**

* Loads the SecurityContext (user's authentication and security information) for the current request.

**7.AuthorizationFilter**

* Checks whether the authenticated user has the necessary permissions to access a resource.

**8.LogoutFilter**

* Handles user logout requests by invalidating the session or token.

**How the Security Filter Chain Works**

1. When a request is received, it goes through the **security filter chain**.
2. Filters are executed in a specific order, determined by Spring Security.
3. Each filter has a specific role, such as:
   * Authenticating the user.
   * Checking if the user has access to a resource.
   * Logging security-related events.

**Order of Filters**

Spring Security filters are arranged in a predefined order. You can view this order in SecurityFilterChain. Here's an example sequence:

1. **ChannelProcessingFilter**(Ensures requests are secure, e.g., HTTPS)
2. **SecurityContextPersistenceFilter**(Loads SecurityContext from session)
3. **HeaderWriterFilter**  
   (Adds security headers, e.g., X-Content-Type-Options)
4. **CsrfFilter**  
   (Validates CSRF tokens)
5. **LogoutFilter**  
   (Processes logout requests)
6. **UsernamePasswordAuthenticationFilter**  
   (Processes form-based login)
7. **BasicAuthenticationFilter**  
   (Processes HTTP Basic authentication)
8. **AuthorizationFilter**  
   (Checks if the user has permissions)
9. **FilterSecurityInterceptor**  
   (Final check for access control)

**6.Session ID**

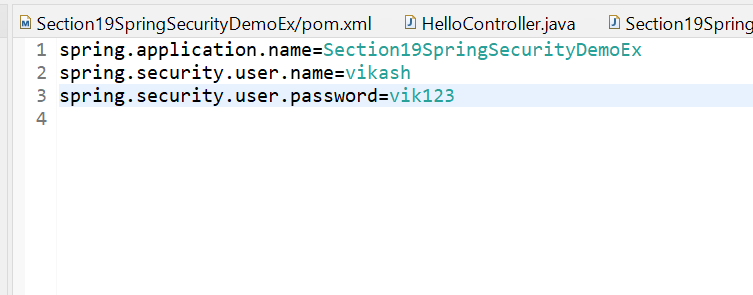
**A Session ID** is a unique identifier used to track a user's session on a website or web application. It is often generated by the server when the user first accesses the site and is stored in the user's browser, typically in the form of a cookie or as part of the URL.

The Session ID helps the server recognize the user during subsequent requests and maintain the user's state across multiple interactions (e.g., keeping the user logged in, tracking items in a shopping cart, etc.). This way, the server can manage information related to that particular session, such as user preferences, login status, or other session-specific data, without the need for the user to authenticate repeatedly.

Key points about Session IDs:

* They are often stored in cookies or URL parameters.
* They ensure that the user's state is preserved as they navigate between pages.
* They are usually short-lived, expiring after a certain period of inactivity or when the user logs out.

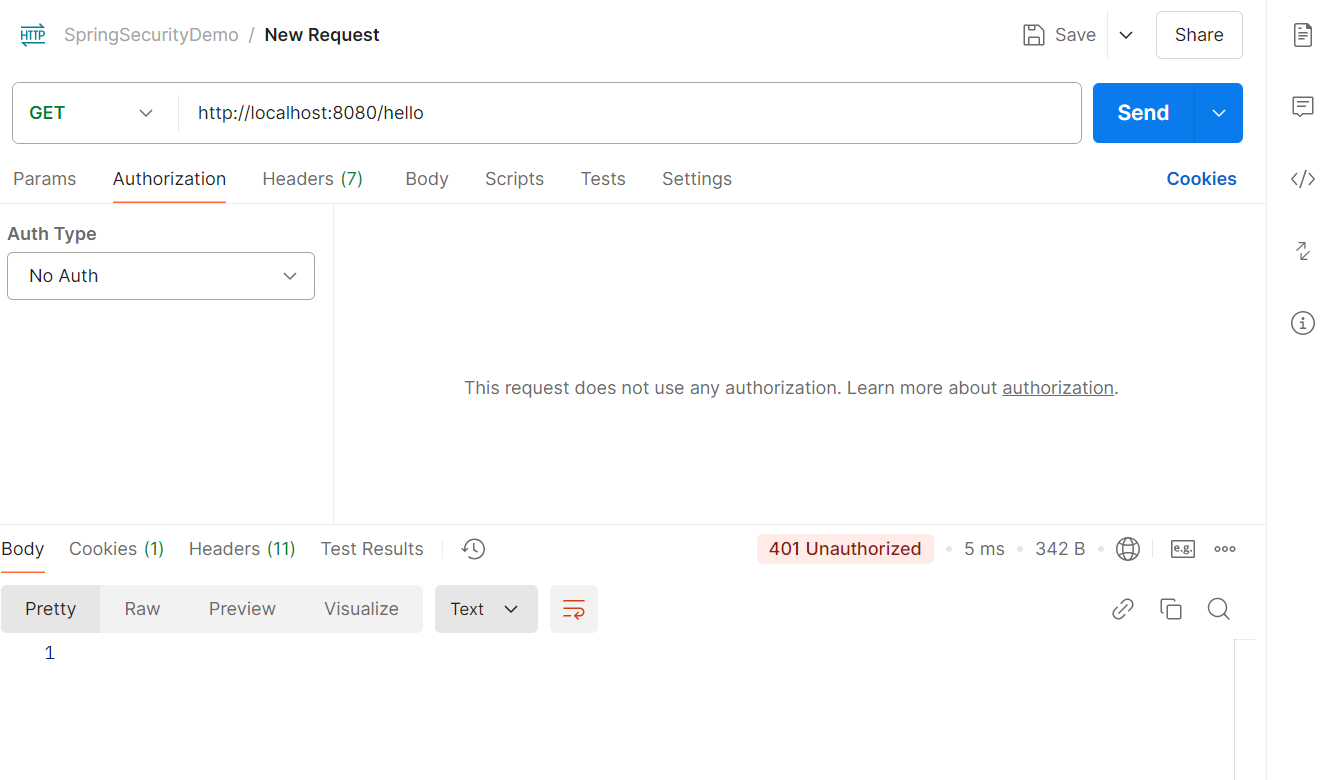
**7.Setting username and password**

****

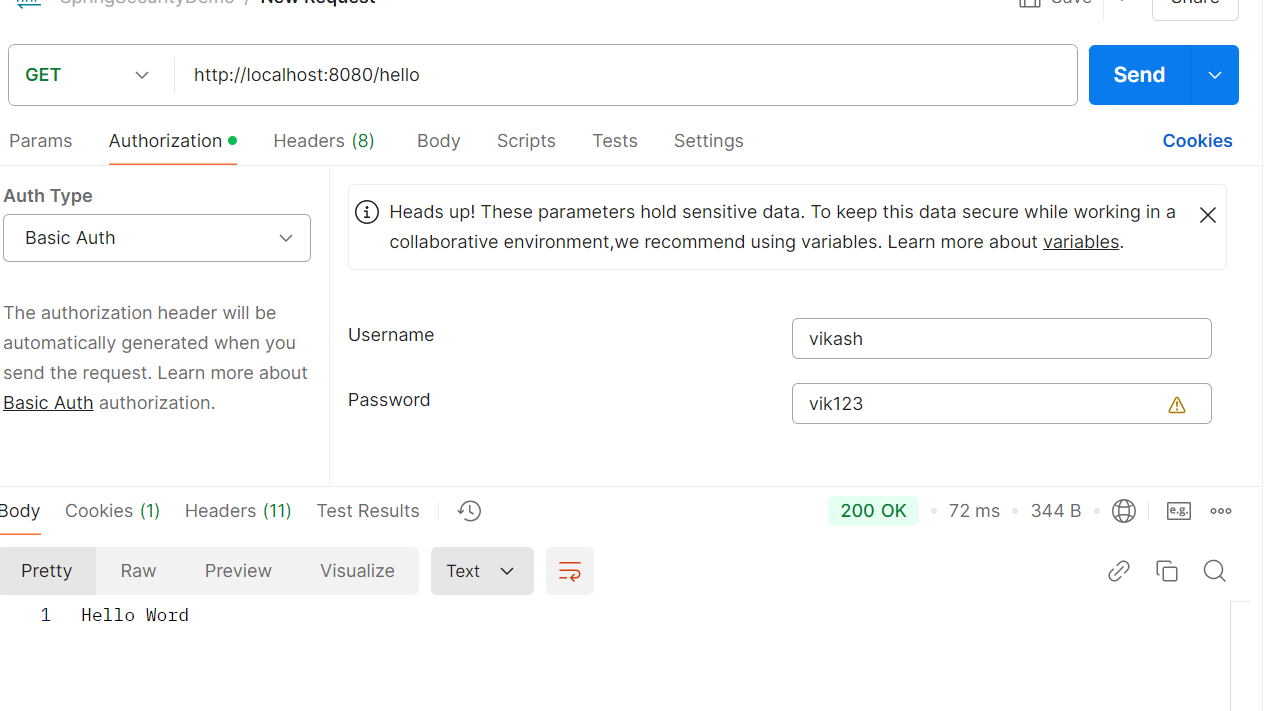
Run the application

Use provided username and password

**8.Basic Auth using Postman**

Without Basic Auth in Postman   


With Basic Auth



**9.What is CSRF**

CSRF is when an attacker tricks a user into performing an action on a website where they are already logged in, without the user's knowledge or consent. The website trusts the user’s session and performs the action thinking it's from the user.

**Step-by-Step Example of CSRF**

**Step 1: User Logs into a Website**

* Imagine you are logged into your **banking website** and you have an active session with them (you’re authenticated).
* The website gives you a **session cookie** in your browser to remember that you're logged in.

**Step 2: Attacker Crafts a Malicious Request**

* The attacker knows that you are logged into your banking site.
* The attacker then creates a **malicious link** that would perform an action on the banking site (e.g., transferring money to the attacker’s account). For example:

html

Copy code

<img src="http://bank.com/transfer?amount=1000&to=attacker\_account">

* This link is a **GET request** that asks the bank to transfer money from your account to the attacker's account. If clicked, it would send a request to the banking site.

**Step 3: The User Clicks on the Malicious Link**

* The attacker sends this link to you via email or a fake website, and you click on it.
* Even though the link looks harmless (like an image or some harmless action), it secretly sends a request to the banking website to transfer money.

**Step 4: Browser Sends the Request with the User’s Session**

* Because you’re **already logged in** to the banking website, your browser automatically includes the **authentication cookie** in the request.

This is what happens:

* + Your browser sends a request to http://bank.com/transfer?amount=1000&to=attacker\_account
  + The request includes your **session cookie** from the banking site, making the bank think it's coming from you.

**Step 5: The Website Processes the Request**

* The banking website sees the request, looks at the session cookie, and **trusts** that the request is from you (since you’re logged in).
* As a result, the banking website **processes the transfer** and sends **1000 units of money to the attacker’s account** without your knowledge!

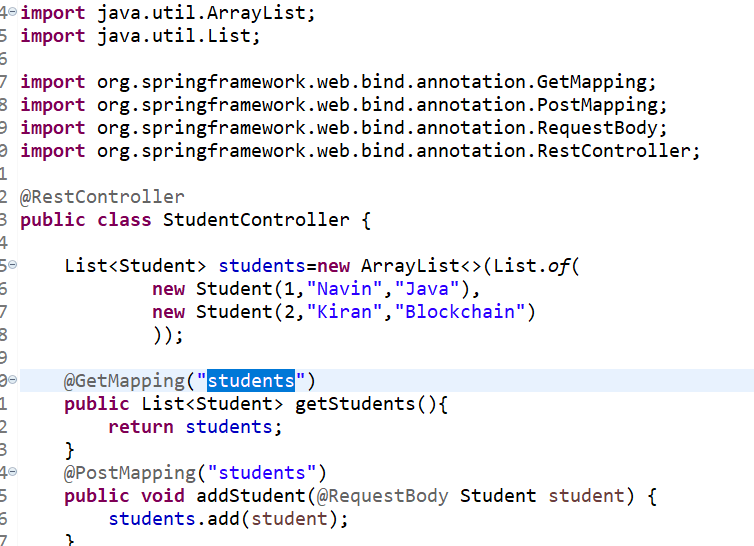
**10.Error without CSRF Token**

**CSRF Example in Spring Boot**

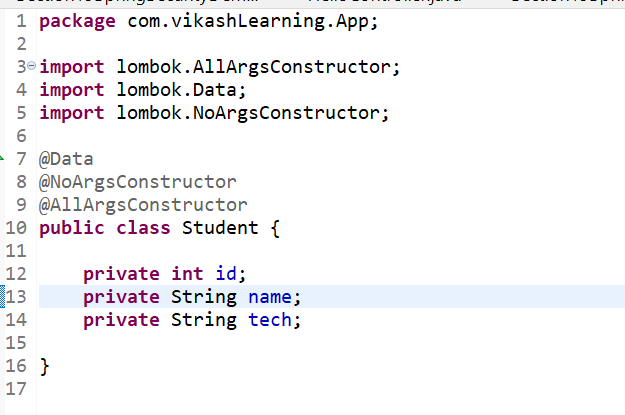
**1. Project Setup**

* You have a Spring Boot application with a StudentController and Student class.
* The StudentController handles the GET and POST requests for managing student data.

1. **StudentController Class**

****

**3.Student.java class**

****

**Explanation:**

* GET request (/students) returns a list of students.
* POST request (/students) allows adding a new student via the request body.

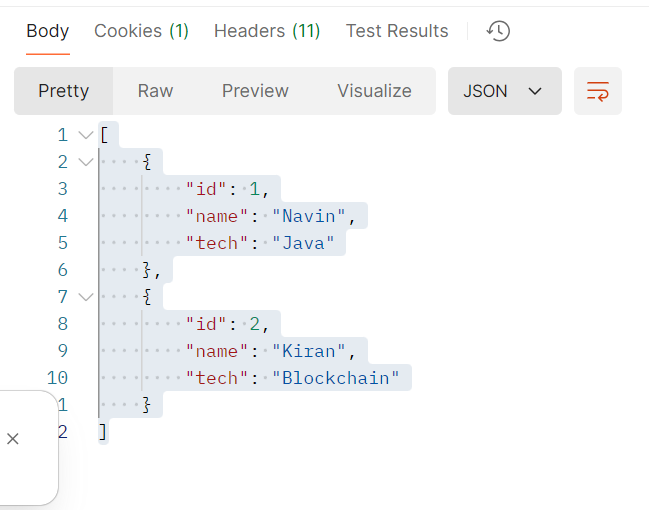
**Explanation**:

* Student is a simple POJO (Plain Old Java Object) representing a student, with fields id, name, and tech (technology).
* Using **Lombok** annotations (@Data, @NoArgsConstructor, @AllArgsConstructor) to generate getter/setter methods and constructors.

1. **Test the Application**

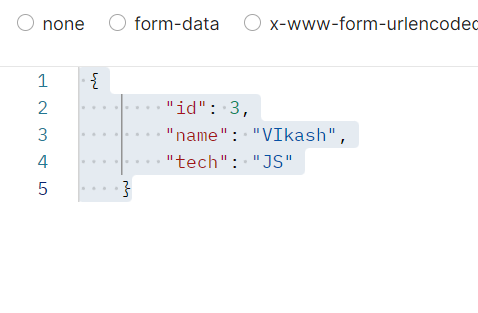
**GET Request:**

* URL: http://localhost:8080/students
* Response (200 OK)

****

**POST Request:**

* **URL**: http://localhost:8080/students
* **Request Body**:



**Response**: 401 Unauthorized

**5. Understanding the 401 Unauthorized Error**

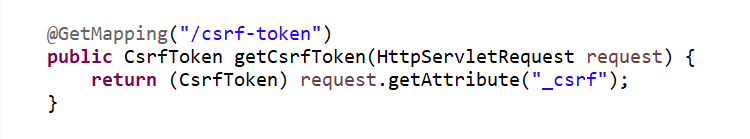
* The error 401 Unauthorized occurs because **authentication is required** to access the POST request, and your application is **not configured to handle this**.
* Spring Security (which is enabled by default in Spring Boot) automatically applies security restrictions on endpoints, including **POST** requests, which require authentication.

**11.Sending CSRF Token**

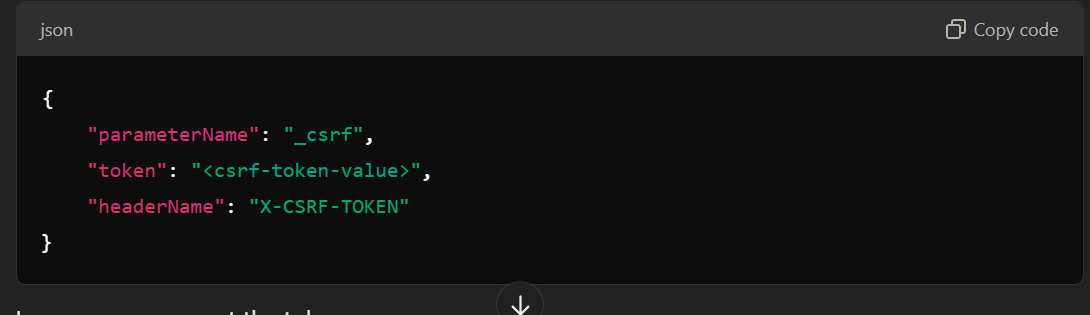
When your Spring application has CSRF protection enabled (which is the default for most Spring Security configurations), the CSRF token must be sent with every state-changing request (like POST, PUT, DELETE). Here’s how you can manage CSRF tokens and use them in your client-side requests.

**1. Retrieve the CSRF Token**

* **What it is:** The CSRF token is a unique value that prevents cross-site request forgery. It is typically sent as part of the response when a session is initiated.
* **Where to get it:** You can retrieve the CSRF token from the HttpServletRequest as shown in your code:



**Response:** When you hit the endpoint /csrf-token, Spring Security provides the CSRF token. The response will look like:



This token should be used in future requests.

**2. Include the CSRF Token in Your POST Request**

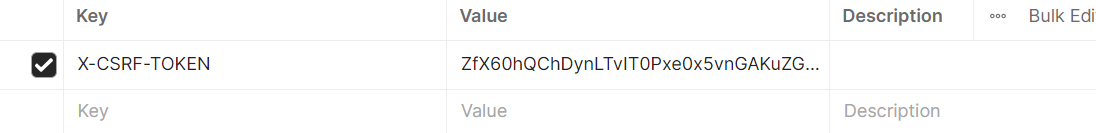
Once you have retrieved the CSRF token, you need to include it in the headers of any subsequent state-changing requests like POST, PUT, DELETE.

* **POST Request to /students with CSRF Token:**
* **URL:** http://localhost:8080/students
* **Method:** POST
* **Headers:**
  + Content-Type: application/json
  + Accept: application/json
  + X-CSRF-TOKEN: <your-csrf-token>

**Example in Postman:**

1. **GET** /csrf-token to retrieve the CSRF token.
2. **POST** /students with the CSRF token in the headers.

**Headers in Postman for POST request:**



Now the previous request will work

**12. Same Site Strict & Security Configuration**

Understanding the SecurityConfig Class

The SecurityConfig class is where you configure your **Spring Security** settings for your application. It includes the necessary configuration to define how the security filters should behave, specifically in terms of authentication, authorization, and CSRF protection.

In your code:



Here’s the breakdown of this code:

* @Configuration: This annotation indicates that this class contains configuration for your Spring application.
* @EnableWebSecurity: This enables Spring Security's web security features. It activates the security framework in your application.
* SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception: This method allows you to configure how HTTP requests are filtered (authentication, authorization, etc.). In this case, http.build() is used to configure the default security setup.

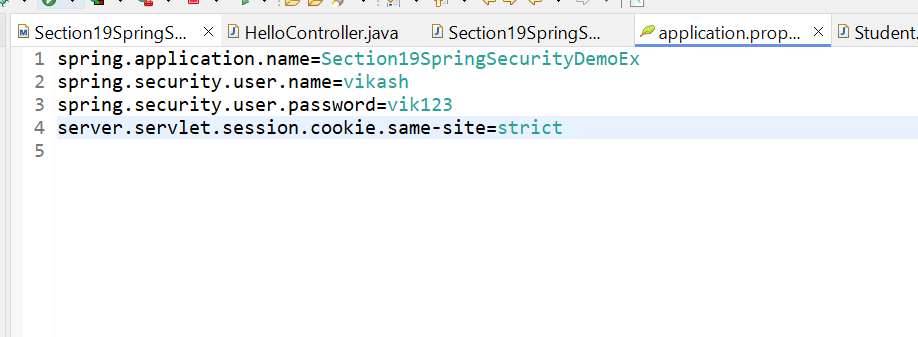
Since you're not adding any specific configuration here, the default security behavior (like HTTP basic authentication, form login, etc.) will be applied.

**2. What is server.servlet.session.cookie.same-site=strict?**

The line in your application.properties file:

properties

Copy code



This line specifies that the cookies used by your application’s **session** will be restricted by the **SameSite** attribute, and it is set to STRICT.

* **SameSite=Strict:** This cookie setting prevents the cookie from being sent along with requests from different sites or domains (cross-origin requests). For example, if your application’s frontend is hosted on a different domain than the backend, **strict** will prevent the cookie from being sent in those cross-origin requests.
  + **Benefits of SameSite=Strict:**
    - More secure against Cross-Site Request Forgery (CSRF) attacks.
    - It ensures that cookies (such as session cookies) are only sent when the user is navigating within the same domain.

**Why use SameSite=Strict?**

* When the SameSite attribute is set to Strict, cookies will only be sent in **same-site** requests. This makes it harder for attackers to perform CSRF attacks since a third-party site won't be able to send a request to your backend with the user's session cookie.
* This setting adds an extra layer of security to prevent unauthorized requests that come from malicious websites.

**Example Scenario:**

Imagine you have a **session cookie** that stores the user’s login state. If a user is logged into your app (say app.com) and they visit another site (say malicious.com), the browser would **not** send the session cookie when the malicious site makes a request to app.com. This is because the **SameSite=Strict** attribute prevents the cookie from being included in the request.

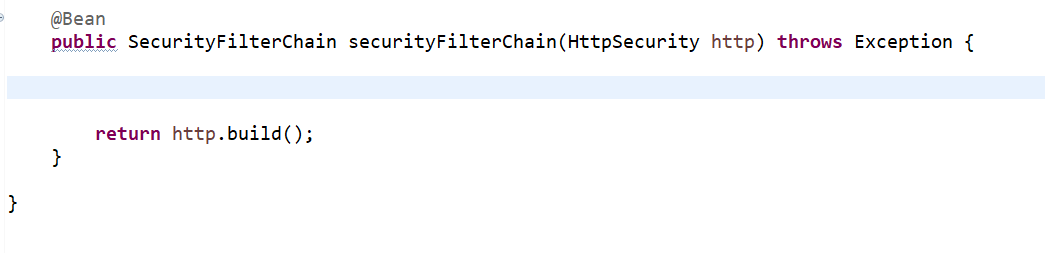
**Question : -- why http://localhost:8080/hello it did not asked me for login after implementin g config class**

The reason your http://localhost:8080/hello endpoint did not prompt for login even after implementing the SecurityConfig class lies in the configuration details (or lack thereof) in your securityFilterChain method.

Here’s what’s happening:

**1. Default Security Behavior**

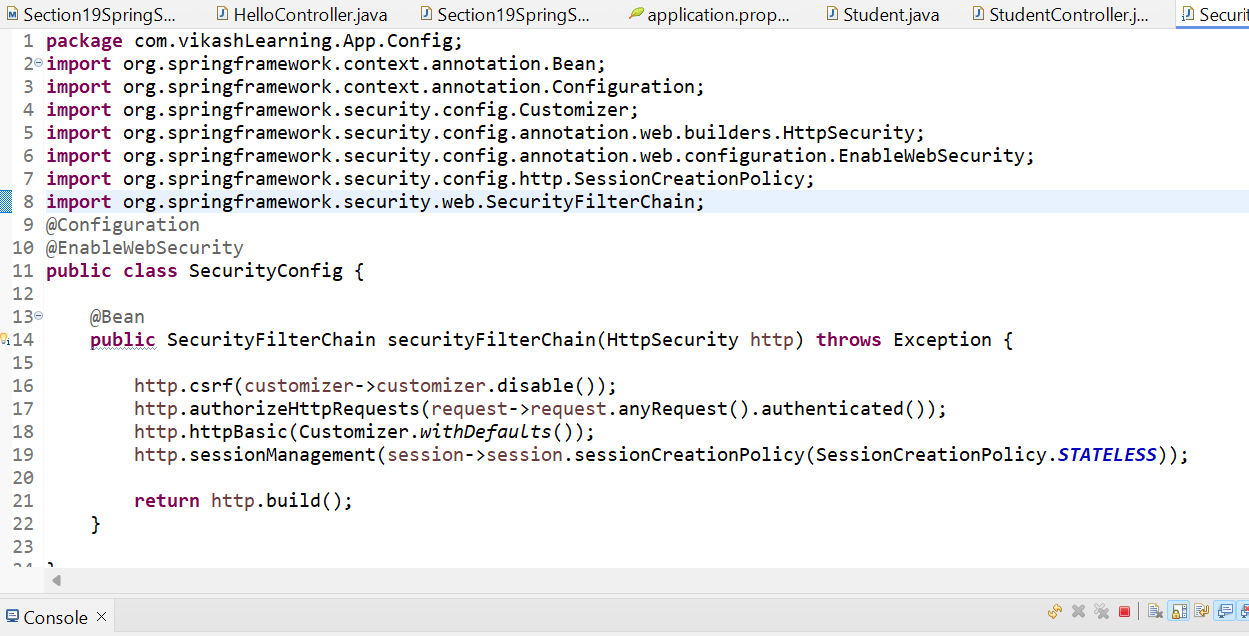
When you define a custom SecurityFilterChain in Spring Security but do not configure specific security rules (like authentication, authorization, or form login), Spring Security will not enforce any security restrictions on endpoints.



This effectively disables Spring Security's default behavior of securing all endpoints. By default:

* Without any configuration, **Spring Security secures all endpoints** and requires authentication for every request.
* By calling http.build() without specifying security rules, you are essentially overriding the defaults, leaving your endpoints unprotected.

**14. Disabling CSRF Token**



**1. http.csrf(customizer -> customizer.disable());**

This **disables CSRF (Cross-Site Request Forgery) protection**.

* **Why disable CSRF?**
  + CSRF protection is useful for applications using **session-based authentication** or cookies to prevent unauthorized actions.
  + In stateless applications, where each request is authenticated independently (e.g., using tokens or Basic Authentication), CSRF protection is unnecessary.
  + Disabling it simplifies API interactions, especially for REST APIs or when using tools like Postman.
* **When should you keep CSRF enabled?**
  + If your application includes forms or uses cookies for authentication, you should **not disable CSRF protection** to prevent attacks.

**2. http.authorizeHttpRequests(request -> request.anyRequest().authenticated());**

This configures **authorization rules** for your application.

* **anyRequest().authenticated()**:
  + This means **all requests** to any endpoint of your application require authentication.
  + Users must provide valid credentials to access any resource.
* **Example:**
  + If you send a request to http://localhost:8080/hello, the server will check whether the user is authenticated before allowing access.
* **Custom Rules (Optional)**: You can define more granular rules if needed, such as:

java

Copy code

http.authorizeHttpRequests(request ->

request

.antMatchers("/public/\*\*").permitAll() // Allow access to "/public/\*\*" without authentication

.anyRequest().authenticated() // All other endpoints require authentication

);

**3. http.httpBasic(Customizer.withDefaults());**

This enables **Basic Authentication**.

* **What is Basic Authentication?**
  + Basic Authentication sends the username and password in the Authorization header of every request.
  + Example of the header:

css

Copy code

Authorization: Basic <Base64Encoded(username:password)>

* **When is Basic Authentication Used?**
  + It’s suitable for simple APIs or when programmatically accessing endpoints (e.g., using Postman or cURL).
  + Not ideal for modern applications with browsers or mobile apps, where token-based authentication (like JWT or OAuth) is preferred.

**4. http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));**

This configures how the application handles user sessions.

* **SessionCreationPolicy.STATELESS**:
  + Indicates that the application **does not maintain sessions** on the server.
  + Each request must be authenticated independently (stateless behavior).
  + There’s no concept of "logged-in" users because the server doesn’t store session state.
* **Why Stateless?**
  + Stateless behavior is essential for REST APIs, where each request is independent and contains all necessary authentication details (e.g., a token in the header).
* **Other Policies:**
  + **SessionCreationPolicy.ALWAYS**: Always create a session, even if one doesn’t exist.
  + **SessionCreationPolicy.NEVER**: Never create a session, but allow access if one exists.
  + **SessionCreationPolicy.IF\_REQUIRED** (default): Create a session only when needed.

**5.How This Works in Practice**

1. **Disabling CSRF**:
   * The application won’t check for CSRF tokens in incoming requests. This makes it easier to test the API using tools like Postman or when using stateless tokens (e.g., JWT).
2. **Authentication Requirement**:
   * All endpoints are protected and require the client to provide valid credentials (username and password) with every request.
3. **Basic Authentication**:
   * Clients include credentials in the Authorization header for each request.

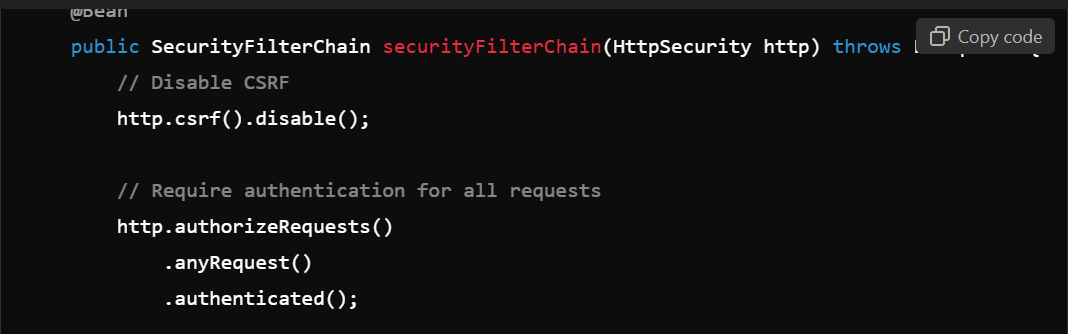
6.**Stateless Sessions**:

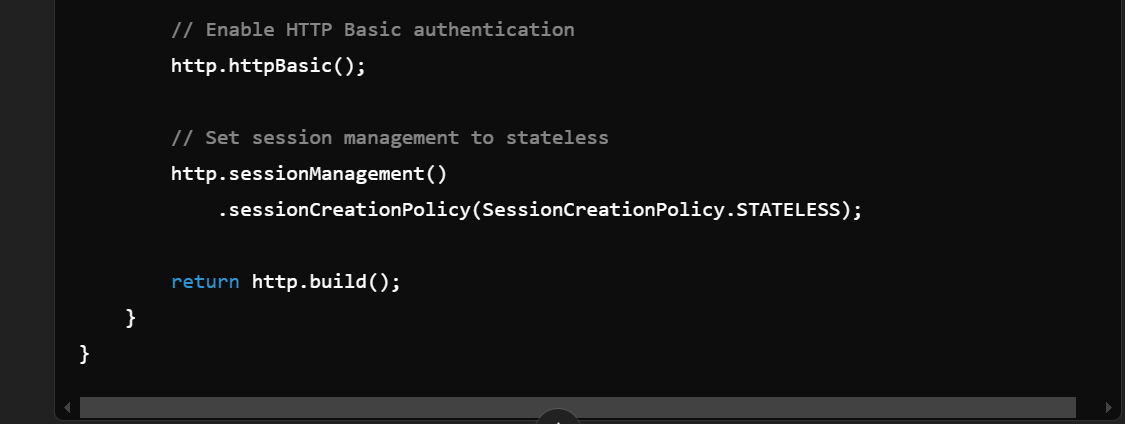
* The application does not create or store sessions. Every request is treated as independent, which is ideal for REST APIs.

**7. Why Use This Configuration?**

* **REST APIs**:
  + Perfect for APIs where each request carries all necessary information.
  + Statelessness improves scalability and simplifies server-side resource management.
* **Simple Authentication**:
  + Basic Authentication is easy to set up and works well for internal APIs or small-scale projects.
* **Security**:
  + Although Basic Authentication is straightforward, it’s less secure unless combined with HTTPS to encrypt credentials.

**15. without lambda**





**Explanation**

**1. Disabling CSRF**

http.csrf().disable();

This disables CSRF (Cross-Site Request Forgery) protection for the entire application.

* **Why disable CSRF?**
  + CSRF protection is typically required for applications using session-based authentication (e.g., forms).
  + For REST APIs (which are usually stateless), CSRF protection is unnecessary because each request is authenticated independently (e.g., using tokens or HTTP Basic).

1. **Authorization**

http.authorizeRequests()

.anyRequest()

.authenticated();

**authorizeRequests()**: Configures the authorization rules.

**anyRequest().authenticated()**: Ensures that all requests to the application require authentication. This means users must provide valid credentials (username and password) to access any resource.

3.**HTTP Basic Authentication**

http.httpBasic();

This enables **Basic Authentication** for the application.

Basic Authentication requires clients to send the username and password in the Authorization header of each request.

1. **Session Management**

http.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

* **What is Session Management?** Configures how the server manages user sessions.
* **SessionCreationPolicy.STATELESS:**
  + Configures the application to **not use server-side sessions**.
  + Each request must provide authentication details, making the application stateless. This is ideal for REST APIs.
  + Stateless sessions improve scalability because the server doesn't need to store session data.

Other options for SessionCreationPolicy:

* ALWAYS: Always create a session.
* NEVER: Never create a session but use an existing one if available.
* IF\_REQUIRED: Create a session only if needed (default behavior).

**How This Works**

1. **CSRF Protection Disabled**:
   * The application doesn’t check for CSRF tokens in requests.
   * Useful for REST APIs, where stateless authentication is used.
2. **Authentication Requirement**:
   * All endpoints are protected and require valid credentials.
   * Example

GET /hello HTTP/1.1

Host: localhost:8080

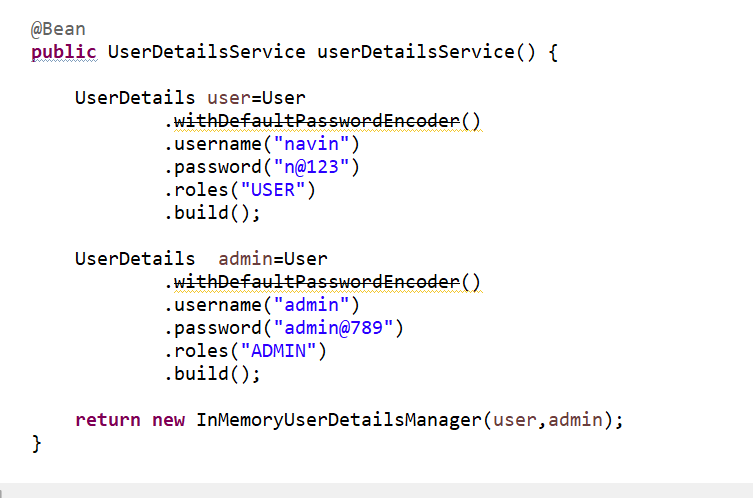
Authorization: Basic dXNlcjpwYXNzd29yZA==

**Stateless Session**:

* The server does not store any session information.
* Each request is treated independently, and authentication must be provided with every request.

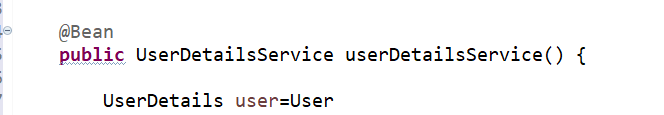
**16.  Getting ready for user database &** **Working with Multiple Users**

****



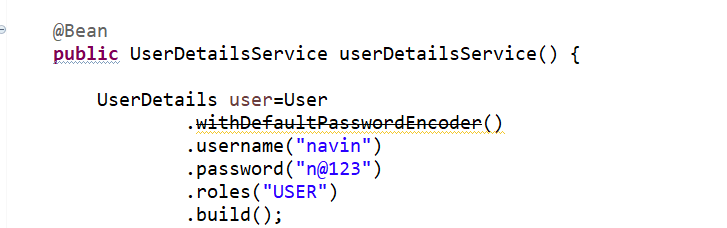
**Code Walkthrough**

**1. Define the UserDetailsService Bean**



* **What is UserDetailsService?**
  + It's a Spring Security interface used to fetch user details (like username, password, and roles) for authentication and authorization.
  + The InMemoryUserDetailsManager is a concrete implementation that stores user details in memory.

1. **Create User Details**

****

**UserDetails:**

* Represents the core information about a user in Spring Security.
* Includes details like username, password, and roles.

**withDefaultPasswordEncoder():**

* Creates a password encoder (uses a simple, insecure hashing mechanism for demonstration purposes).
* Important: Do not use this in production! Instead, use a stronger encoder like BCryptPasswordEncoder.

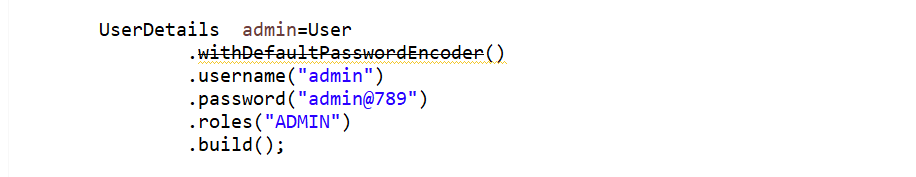
**username("navin") and password("n@123"):**

* Set the username and password for this user.

**roles("USER"):**

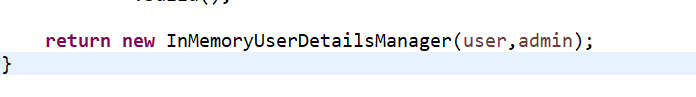
* Assigns the user a role (USER). Spring automatically prefixes roles with ROLE\_.

1. **Create Admin Details**



* Same as the user configuration, but this user has the role of ADMIN.

1. **Store the Users in Memory**



* **InMemoryUserDetailsManager**:
  + A built-in implementation of UserDetailsService that stores user details in memory.
  + It's suitable for simple applications or testing purposes.
  + For production, use a database-backed implementation.

**How This Works**

1. **When a User Tries to Authenticate:**
   * Spring Security fetches the user details using the UserDetailsService.
   * In this case, it retrieves details from the in-memory store.
2. **When Authentication Succeeds:**
   * The user can access resources based on their role (USER or ADMIN).

**Question :What is use withDefaultPasswordEncoder() what does it mean**

The method withDefaultPasswordEncoder() in Spring Security is a utility method to quickly create a UserDetails object with a simple, built-in password encoder. Here's a detailed explanation:

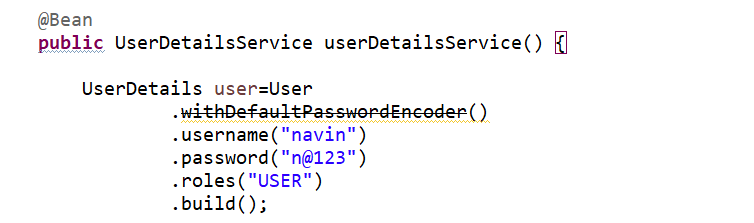
**Purpose**

* It is a convenience method to create users during development or testing without needing to manually configure a PasswordEncoder.
* It internally uses a basic, insecure password encoding mechanism for simplicity

**How It Works**

* **Encoding Mechanism**:
  + The method uses a **delegating password encoder**, which defaults to NoOpPasswordEncoder for encoding passwords.
  + It prefixes passwords with {noop}, which indicates to Spring Security that the password is stored in plain text and no hashing is applied.

**Example**:



In the above example, the password "n@123" will be stored as {noop}n@123.

**Why Use withDefaultPasswordEncoder()?**

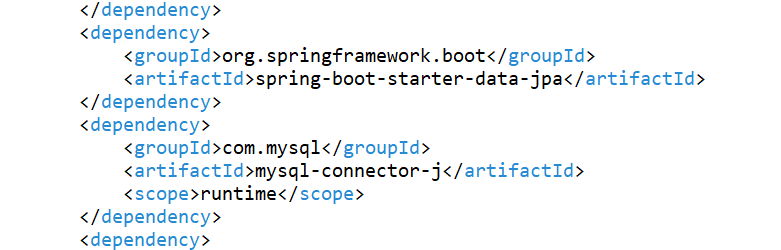
* **For Quick Setup**:
  + It allows you to quickly create users without worrying about configuring a PasswordEncoder.
* **For Learning or Prototyping**:
  + It's helpful when you're learning or setting up a prototype application.

**Why Not Use It in Production?**

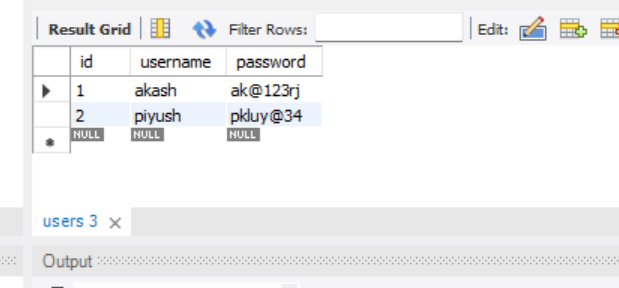
* The password is stored in plain text (e.g., {noop}password).
* It doesn't provide the security required for production environments, where passwords should always be securely hashed.

**18. Creating User table and db properties**

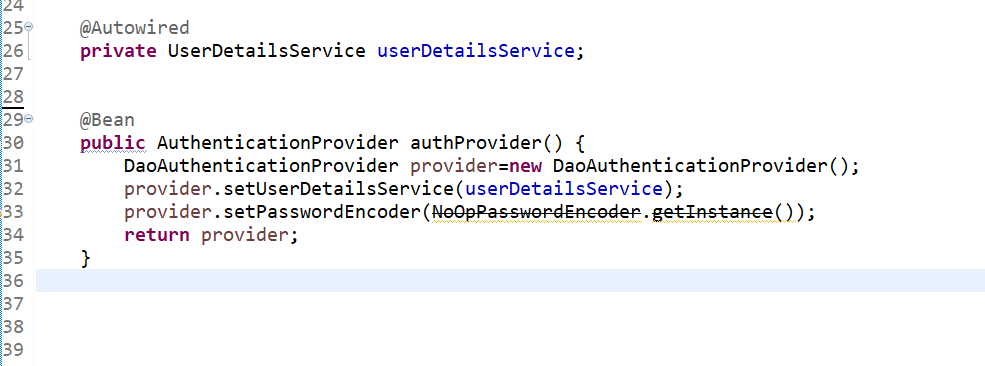
Add below dependencies



Create table and insert the data

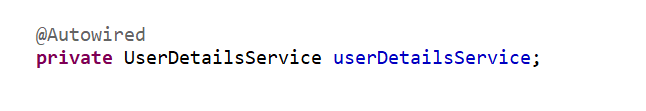


**19.  AuthenticationProvider**



Code Breakdown and Explanation

1.Injecting UserDetailsService



**Purpose**: You are injecting a UserDetailsService bean to fetch user details (username, password, roles) from a data source (e.g., in-memory, database, etc.).

This is essential when you want Spring Security to rely on your custom logic to retrieve user details for authentication instead of the default auto-configured user.

**What is UserDetailsService in Spring Security?**

UserDetailsService is a core interface in Spring Security that is responsible for retrieving user-related data. It is used to authenticate a user by loading user-specific data (like username, password, and roles/authorities) from a data source, such as a database, LDAP, or even an in-memory list.

**Key Features of UserDetailsService**

1. **Primary Use**:
   * It provides the data that Spring Security requires to authenticate and authorize a user.
2. **Single Method**: The UserDetailsService interface has only one method:

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

**username**: The username provided by the user during login.

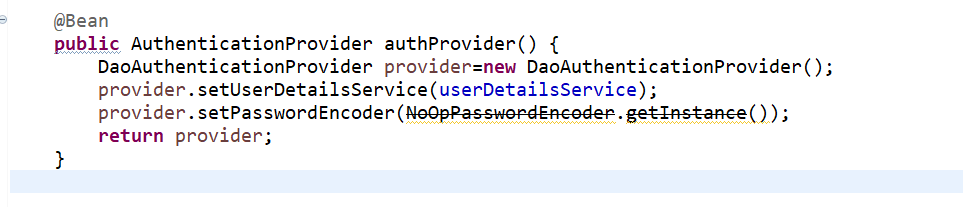
**Returns**: A UserDetails object containing information about the user.

**Throws**: A UsernameNotFoundException if the user with the given username is not found.

**Customizable**:

* You can implement this interface to customize the way user data is retrieved. For instance, you can fetch user data from a database, an external service, or any other source.

**2.Defining a Custom Authentication Provider**

****

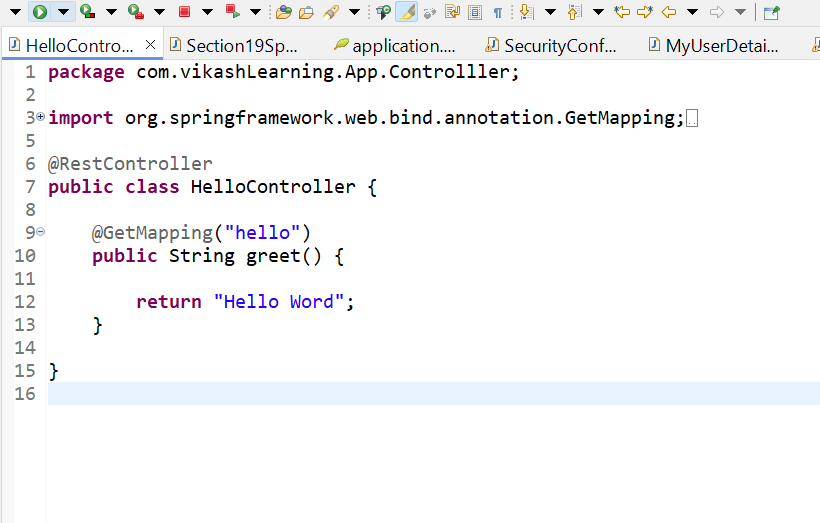
**What It Does:**

* DaoAuthenticationProvider: A specific implementation of AuthenticationProvider that uses a UserDetailsService to authenticate the user.
* setUserDetailsService(userDetailsService): Associates the DaoAuthenticationProvider with your custom UserDetailsService for retrieving user details.
* setPasswordEncoder(NoOpPasswordEncoder.getInstance()): Configures the provider to use no encoding for the password, meaning passwords are stored and compared as plain text (not recommended for production).

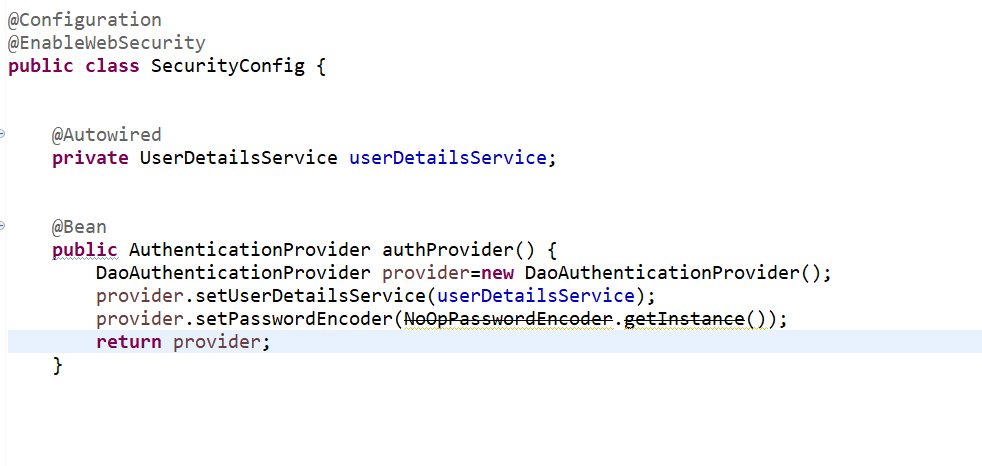
**Why It's Needed:**

* Spring Security, by default, uses an auto-configured AuthenticationProvider. By creating your own, you override this behavior to control how users are authenticated and passwords are verified.
* Without this custom provider, your injected UserDetailsService would not be utilized in the authentication process.

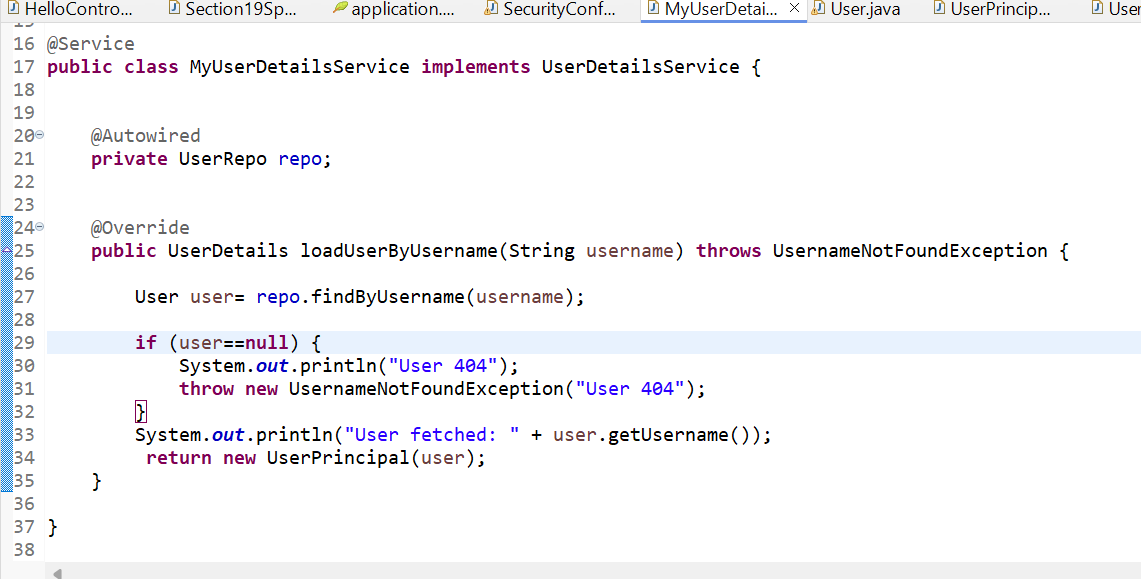
**20. Creating a UserDetailsService & User Repository & UserDetails and UserPrincipal**

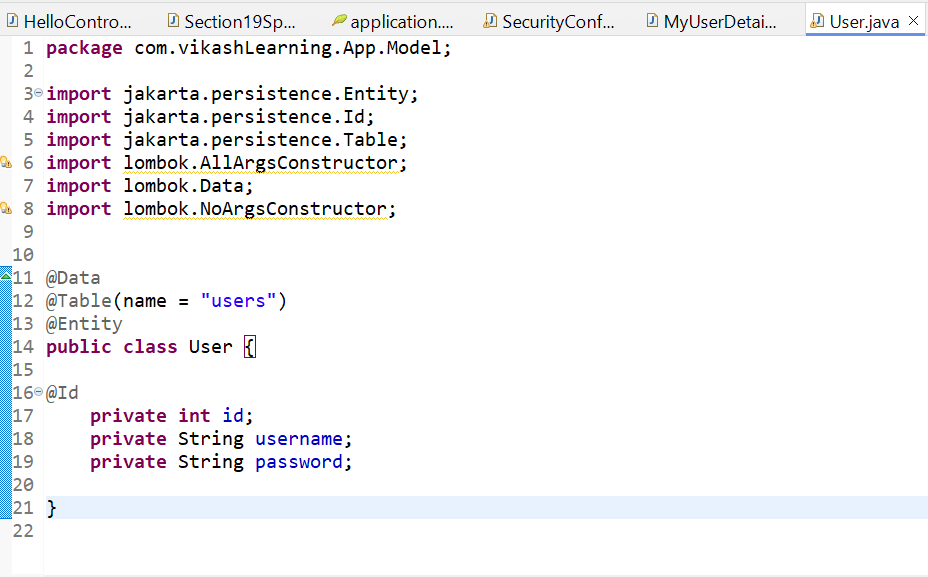
Find the all code here   
  
**Hello.controller**  


This is SecurityConfig class

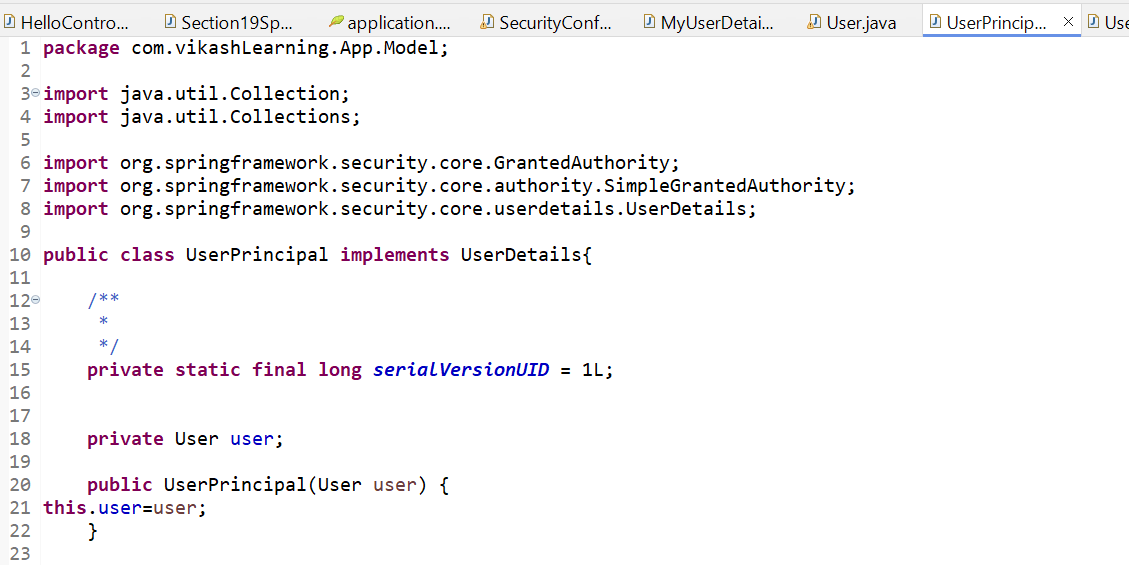
****

****

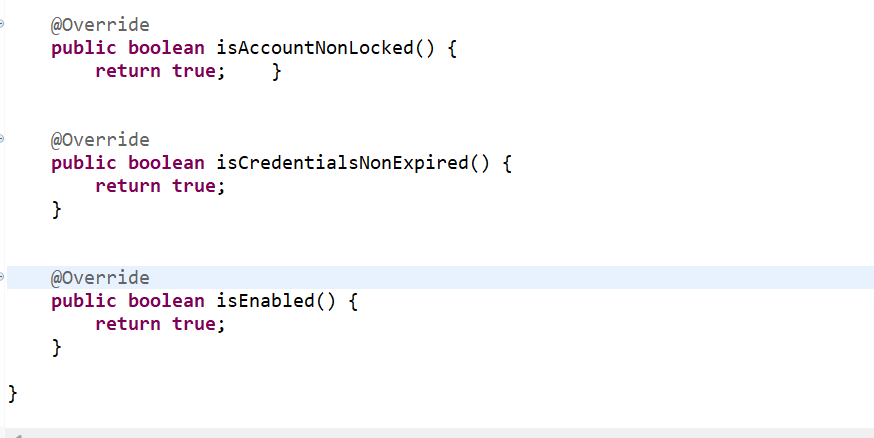
**This is Service class   
  
**

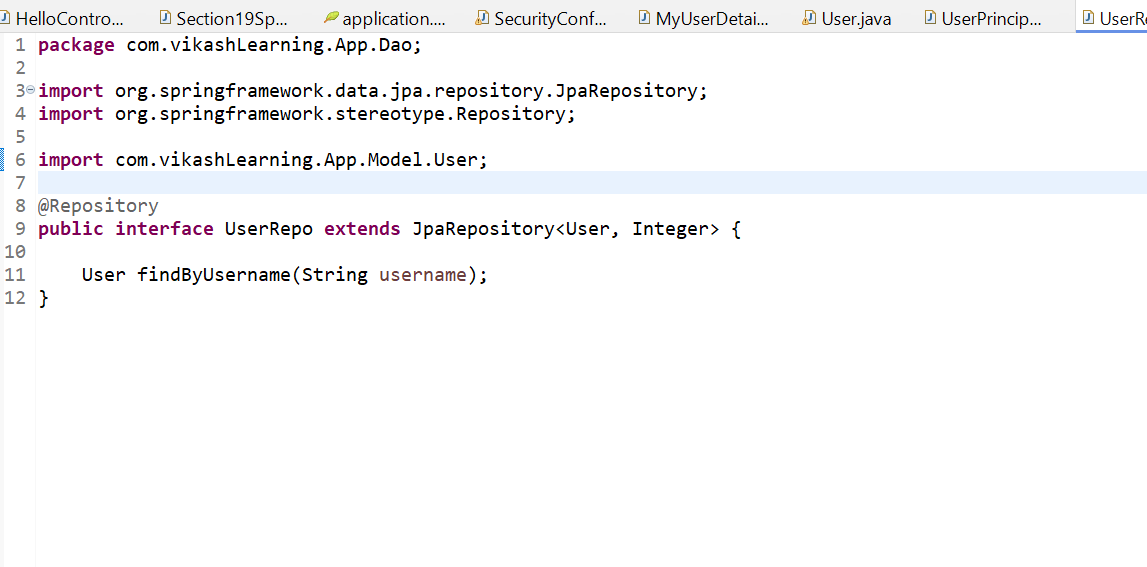
**This is model calss  
  
**

**This is USerPrinciple class**

****

****

****

**This is user repo class   
  
**

**Complete Explanation of Class Interactions in Hierarchical Order**

Let's explain the flow of how all classes interact in a step-by-step manner. Each explanation will start with the respective class and will follow the execution hierarchy. A flowchart will be added at the end for clarity.

1. **HelloController (Entry Point)**

* **Purpose**:  
  Handles the incoming requests to the application.
* **Key Method:**@GetMapping("hello"):
  + Maps the /hello URL to the greet() method.
  + Simply returns the string "Hello World" for authenticated users.
* **Interaction**:
  + Relies on Spring Security for authentication before processing the request.
  + If authentication succeeds, this controller processes the request.

2**. SecurityConfig (Spring Security Configuration)**

* **Purpose**:  
  Configures the security of the application, including authentication mechanisms and session management.
* **Key Beans and Methods**:
  1. **AuthenticationProvider Bean**:
     + **Type**: DaoAuthenticationProvider.
     + **Purpose**: Used to authenticate users by delegating the user retrieval to the UserDetailsService.
     + **Interaction**:
       - Calls MyUserDetailsService (injected as userDetailsService) to fetch user details.
       - Uses NoOpPasswordEncoder to compare the passwords

**provider.setUserDetailsService(userDetailsService);**

**Explanation**:

* userDetailsService is an instance of MyUserDetailsService.
* Fetches user details from the database through UserRepo.
  1. **SecurityFilterChain Bean:**
     + **Configures HTTP security:**
       - Disables CSRF.
       - Ensures all requests are authenticated.
       - Enables stateless sessions.
* **Interaction**:
  1. Ensures all incoming requests go through the authentication flow provided by Spring Security.

3. **MyUserDetailsService (Custom Implementation of UserDetailsService)**

* **Purpose**:  
  Fetches user details from the database for authentication.
* **Key Method:**loadUserByUsername(String username):
  + **Input**:  
    username (provided in the login request).
  + **Process**:
    - Calls UserRepo.findByUsername(username) to query the users table.
    - If no user is found, throws UsernameNotFoundException.
    - If a user is found, converts the User entity to UserPrincipal and returns it.
  + **Output**:  
    UserPrincipal object, which implements UserDetails.
* **Interaction**:
  + Interacts with UserRepo to query the database.
  + Converts User entity to UserPrincipal to supply to DaoAuthenticationProvider.

**4. UserRepo (Repository Layer)**

* **Purpose**:  
  Acts as the data access layer to interact with the database.
* **Key Method:**findByUsername(String username):
  + Executes a query on the users table to fetch a user with the given username.
* **Interaction**:
  + Used by MyUserDetailsService to fetch user data from the database.
  + Returns a User entity if found, or null if no match is found.

5**. User (Model Class)**

* **Purpose**:  
  Represents the users table in the database.
* **Structure**:  
  Contains fields like id, username, and password.
* **Interaction**:
  + Fetched by UserRepo from the database.
  + Wrapped by UserPrincipal to supply to Spring Security.

6**. UserPrincipal (Custom UserDetails Implementation)**

* **Purpose**:  
  Adapts the User entity to the UserDetails interface required by Spring Security.
* **Key Methods:**
  + getUsername(): Returns the username.
  + getPassword(): Returns the password.
  + getAuthorities(): Provides a default authority of USER.
* **Interaction:**
  + Receives the User entity from MyUserDetailsService.
  + Supplies the username, password, and roles to DaoAuthenticationProvider for authentication.

**End-to-End Flow**

1. User Sends a Login Request:
   * User provides username and password.
2. Spring Security Handles the Request:
   * Delegates the request to DaoAuthenticationProvider for authentication.
3. DaoAuthenticationProvider Fetches User Details:
   * Calls MyUserDetailsService.loadUserByUsername(username).
4. MyUserDetailsService Queries the Database:
   * Calls UserRepo.findByUsername(username) to fetch the user.
5. Database Query Execution:
   * UserRepo retrieves the User entity from the database.
6. Wrap User into UserPrincipal:
   * MyUserDetailsService wraps the User entity in a UserPrincipal object.
7. Password Verification:
   * DaoAuthenticationProvider compares the provided password with the stored password using NoOpPasswordEncoder.
8. Authentication Success or Failure:
   * If the passwords match, the user is authenticated.
   * If not, an exception is thrown.

**Flowchart**

User Request

↓

Spring Security

↓

DaoAuthenticationProvider

↓

MyUserDetailsService

↓

UserRepo

↓

Database (Fetch User)

↓

UserPrincipal (Wrap User Entity)

↓

DaoAuthenticationProvider (Password Validation)

↓

Authentication Result

↓

HelloController (If Authenticated)

**1. User submits the username and password**

When the user sends the request (e.g., via HTTP Basic Authentication), they provide their username and password (usually in the Authorization header). This is handled by Spring Security.

**2. Spring Security Authentication Process**

* Spring Security uses the DaoAuthenticationProvider to manage authentication. When you configured DaoAuthenticationProvider, you set the UserDetailsService (your MyUserDetailsService) and the password encoder (NoOpPasswordEncoder).
* The provider (DaoAuthenticationProvider) manages the authentication process but doesn't call loadUserByUsername directly. Spring Security calls loadUserByUsername for you when the DaoAuthenticationProvider needs to load the user details.

**3. The loadUserByUsername method**

* The DaoAuthenticationProvider calls the loadUserByUsername method in your MyUserDetailsService to fetch the user details based on the username.
* In the loadUserByUsername method:
  + You use the repo.findByUsername(username) method to fetch the user from the database by their username.

**java**

**Copy code**

**User user = repo.findByUsername(username);**

At this point, the user object is fetched from the database, and it contains both the username and the password (the password that is stored in your users table).

**4. Password verification**

* After retrieving the user object from the database, Spring Security performs password verification.
* Spring Security compares the password entered by the user (from the HTTP request) with the password fetched from the database. The password from the request is passed automatically by Spring Security as part of the authentication process.

Since you are using NoOpPasswordEncoder, it does not perform any encryption; it simply compares the plaintext password from the request with the password stored in the database.

**Where does password comparison happen?**

* The password comparison happens within the DaoAuthenticationProvider after it calls loadUserByUsername and retrieves the User object from the database.
* The DaoAuthenticationProvider is responsible for validating the password by comparing the one provided in the authentication request with the one retrieved from the database.

**To summarize the process:**

1. The user enters their username and password in the request.
2. Spring Security's DaoAuthenticationProvider is used to handle authentication.
3. The DaoAuthenticationProvider calls loadUserByUsername (your method in MyUserDetailsService).
4. Inside loadUserByUsername, the repo.findByUsername(username) method fetches the user from the database.
5. The password entered by the user is compared with the password fetched from the database by Spring Security (this happens internally in the DaoAuthenticationProvider).
6. If the passwords match, authentication is successful.

The password comparison and verification do not happen directly inside loadUserByUsername; they happen after the UserDetails (i.e., the UserPrincipal) is returned to DaoAuthenticationProvider, which performs the final authentication checks.

**23.  What is Bcrypt**

* **BCrypt** is a password hashing algorithm that provides **strong security** for storing passwords.
* It uses a **salt** to protect against rainbow table attacks (attacks using precomputed hashes).
* BCrypt is **adaptive**: as computing power increases over time, you can increase the complexity factor, making it harder to brute-force passwords.
* The BCrypt password hashing process creates a hashed password, which is stored in the database, and not the raw password.

The important features of BCrypt are:

* It **hashes** the password in a way that’s slow enough to deter brute-force attacks.
* The **salt** is randomly generated for each password, ensuring unique hashes even for identical passwords.

**24. User Registration & BCrypt Encoding for User Registration & . Setting Password Encoder**

In your UserController and UserService, you’ve added functionality for password encoding using BCrypt when a user registers.

* In your UserService, you’ve added the following line to encode the password before saving the user:

user.setPassword(encoder.encode(user.getPassword()));

**Explanation**:

* + **encoder.encode(user.getPassword())**: This encodes the plain-text password using BCrypt before it is stored in the database. This ensures that you don't store plain-text passwords.
  + **BCryptPasswordEncoder(12)**: The argument 12 specifies the strength or complexity of the hash. The higher the number, the more time-consuming the hashing process, making it more resistant to brute-force attacks.

Now, the password will be stored in the database as a hashed value, rather than as plain text.

You are then saving the **encoded password** to the database:

return repo.save(user);

This saves the user object with the hashed password.

**SecurityConfig Changes (Password Encoding)**

You’ve updated the SecurityConfig class to use BCryptPasswordEncoder in the AuthenticationProvider

provider.setPasswordEncoder(new BCryptPasswordEncoder(12));

**Explanation**:

* **BCryptPasswordEncoder(12)**: Here, you’ve updated the password encoder used for authentication to BCryptPasswordEncoder. This ensures that Spring Security knows how to **compare the encoded password** (from the database) with the password entered by the user during login.
* When a user tries to log in, Spring Security will compare the hashed password stored in the database (using BCrypt) with the password the user provides. The BCrypt encoder will internally handle the comparison and check if the passwords match.

**4. Summary of the Changes**

* In your **UserService**:
  + The password entered during user registration is encoded using BCrypt before being stored in the database.
* In your **SecurityConfig**:
  + The BCrypt encoder is used for password verification during authentication (to compare the provided password with the stored hash).

**How BCrypt Works:**

1. When a user registers, their password is **hashed** with a unique salt and stored in the database.
2. When a user logs in, Spring Security uses the **BCryptPasswordEncoder** to hash the entered password and compare it with the hash stored in the database.
3. If the hashed values match, the authentication is successful.